THE QASAR
DUAL PROCESSOR
MICROCOMPUTER

OPERATION MANUAL

DECEMBER, 1980

15 Boundary Street
Rushcutters Bay
SYDNEY, AUSTRALIA 2011
Telephone (02) 335222
Telex AA27998

TABLE OF CONTENTS Page

1.0 THE QASAR COMPUTER 1
1.1 Memory Organization 1
1.2 Firmware 2
1.3 Dual Processor Operation 7
1.4 Using The 6800/6809 System 7
1.5 Interfacing 9
1.6 Floppy Disk Operations 9
1.7 Front Panel Controls 10
1.8 System Startup Procedure 12
1.9 Interrupts 12
1.9.1 Using Interrupts with the Dual 6800 . . . 13
1.9.2 Using Interrupts with the 6800/6809 . . . 14

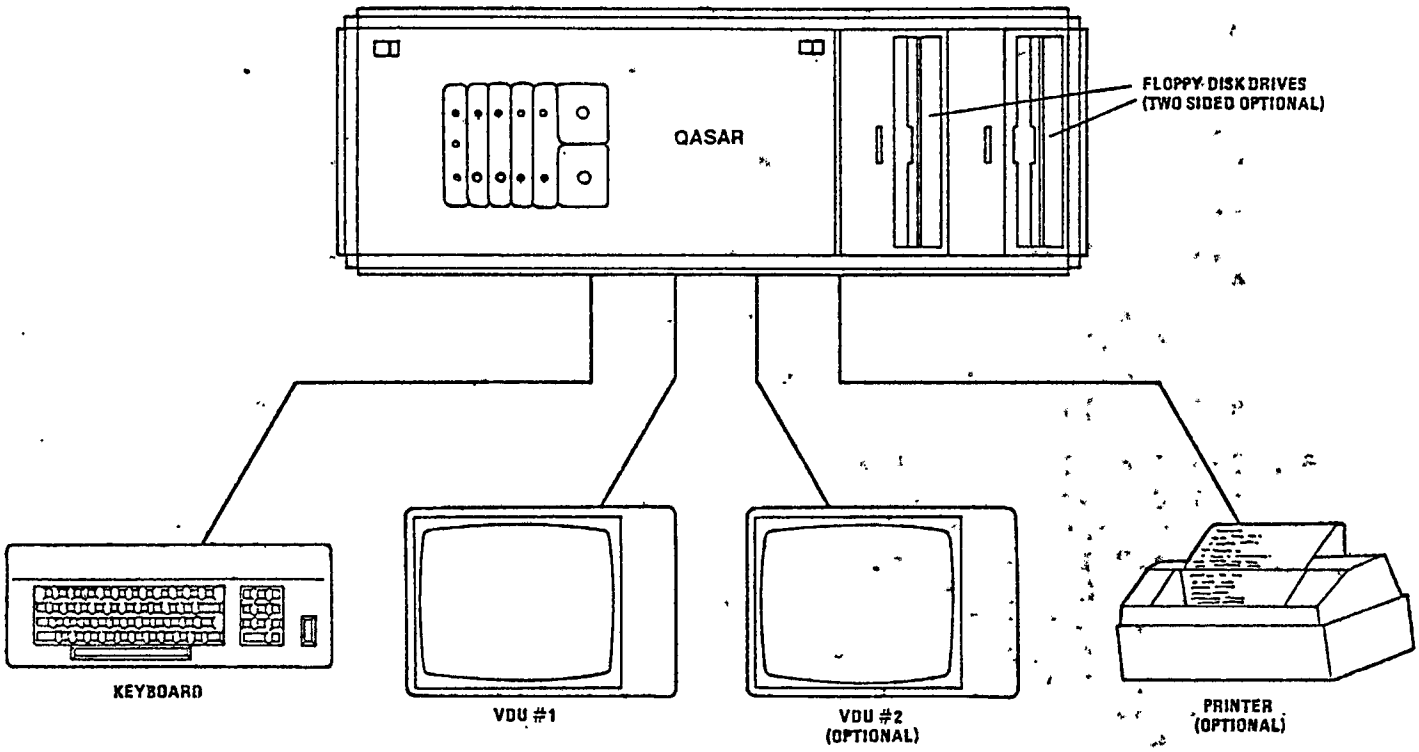
2.0 THE QDOS OPERATING SYSTEM 15
2.1 QDOS Memory Restrictions 16
2.2 Writing and Running a Program with QDOS 16
2.3 Program Debugging 19

3.0 QASAR GRAPHICS / LIGHT-PEN SYSTEM 20
3.0.1 System requirements 20
3.1 Loading the ICPACK 20
3.1.1 Loading by command 21
3.1.2 Auto-load by QDOS 21
3.1.3 When not using QDOS 21
3.2 Keyboard 22
3.3 Line Printer 22
3.4 Video Screen 22
3.5 Character output 23
3.6 Escape sequences 23
3.7 Pen versus cursor 23
3.8 Commands by group 24
3.9 Screen colour 24
3.9.1 BC n Set background colour 24
3.9.2 IN Invert screen image 24
3.10 Graphics group 25
3.10.1 Pen colour 26
3.10.2 PC n - Set pen colour. 26
3.10.3 Offset register 26
3.10.4 OF x,y - Set offset to x,y 26
3.10.5 OR x,y - Add x,y to offset. 26
3.10.6 Move pen 26
3.10.7 MA x,y - Move pen absolute. 26
3.10.8 MR x,y - Move pen relative. 26
3.10.9 Vector plot 27
3.10.10 PA x,y - Plot absolute. 27
3.10.11 PR x,y - Plot relative. 27
3.10.12 Dot (point) plotting 27
3.10.13 DA x,y - Plot a dot absolute. 27
3.10.14 DR x,y - Plot a dot relative. 27
3.11 Character group 28
3.11.1 Cursor Mode 29
3.11.2 CU n - Set cursor mode. 29
3.11.3 Cursor movement 29
3.11.4 MC l,c - Move cursor to line,column. . . 29

3.11.5 CP - Move cursor to pen position. . . . 29
3.12 Scrolling area 30
3.12.1 TS 1 - Set top of scroll 30
3.13 Protected Fields 30
3.13.1 PF line,char,len - Define a field. . . . 30
3.13.2 GF field - Move cursor to a field. . . . 30
3.14 Light pen group 31
3.14.1 Light-pen commands 31
3.14.2 LG m - Return position in form x,y. . . 31
3.14.3 LC m - Return line,char. position . . . 31
3.14.4 LF m - Return a field number. 31
3.15 Defaults 32
3.16 Screen Control Codes 33
3.17 Graphics Command Summary 34
3.18 Examples of Use of Graphics 35
3.19 QASAR Compiler Basic 35

4.0 QASAR ROM SUBROUTINES 36
4.1 Peripheral I/O Functions 37
4.2 PROCESSOR 1 SPECIAL ROUTINES 41
4.2.1 Time-of-day clock 41
4.2.2 User Timer 42
4.2.3 Keyboard Character Flag 42
4.3 Floppy Disk Controller Subroutines 43

5.0 DEBUG MONITOR PROGRAM (MONTR) 47
5.1 Command Prompt 48
5.2 Opening commands 48
5.2.1 The Slash, / 48
5.2.2 The Reverse Slash, \ (shift-L) 49
5.2.3 The Line Feed, <line feed> (control-J) . 49
5.2.4 The Up Arrow, <up arrow> (shift-N) . . . 50
5.2.5 Named Locations, $ 50
5.3 Changing the Contents of a Location 51
5.3.1 The Return, <return> 52
5.3.2 Mixed Open, Modify and Close commands . . 53
5.4 Address Sequence Operations 53
5.5.1 The "AT" Symbol, @ (shift-P) 54
5.5.2 The Right Angle Bracket, > 54
5.5.3 The Left Angle Bracket, < 55
5.5.4 Branch Offset Calculation, ;O 55
5.5 Breakpoint Facilities 57
5.5.1 Setting Breakpoints 57
5.5.2 Removing Breakpoints 58
5.5.3 Examining Breakpoints 58
5.6 Running A Program 59
5.6.1 The GO command, ;G 59
5.6.2 The Proceed command, ;P 59
5.7 The FILL command, ;F 60
5.8 Address Relocation Techniques 61
5.8.1 The Segment Relocation Command, ... 61
5.8.2 The Index Relocation Command, , ... 62
5.9 MONTR Command Summary 63



TYPICAL QASAR COMPUTER SYSTEM

-SECTION 1-

1.Ø THE QASAR COMPUTER

The QASAR dual processor is designed as a general purpose computer suitable for commercial and scientific data processing, program development, data gathering, and many other applications. An extensive range of software is available for the QASAR, including a disk operating system, editors, assemblers, and high level language compilers.

The heart of the QASAR is its powerful Dual Processor architecture. Two microprocessors operate effectively simultaneously on the one common bus. The inclusion of a second microprocessor adds very little to the cost of the computer yet allows substantial speed improvements and program simplification in many applications. For instance, a system which would require a lot of interrupt handling with a single processor can be implemented more simply using a dual processor with one CPU dedicated to a polling and buffering loop, allowing the main processor to handle complete messages rather than separate data items. The QASAR uses this technique for all non-disk I/O.

The QASAR'S modular construction allows easy expansion or modification to suit a user's specific requirements. The basic computer consists of a power supply, two single or double sided floppy disk drives, a card cage, backplane, and chassis. Two processor cards are available- a dual 6800 card and a 6800/6809 version. A typical system would also include a processor control card, 64K memory card, floppy disk controller, and dual TVT interface (memory mapped video display). Other options include a graphics display card and a light pen controller.

The minimum peripheral requirement for the computer is a keyboard and video monitor, but a second monitor and printer may be added. A standard serially interfaced computer terminal is not normally required but may be used in place of the keyboard and TVT.

1.1 Memory Organization

The QASAR system uses two processors which talk to a common bus, both running at their maximum rated clock speed of 1 MHz. Data and address signals for each processor are interleaved, resulting in a 2 MHz bus. All QASAR RAM cards have a cycle time of 500 ns, thus each processor can have it's own independent address space without causing bus contention. Blocks of memory can be assigned as common to both processors, allowing inter-processor communication.

The top 4K of memory space ($F000-$FFFF) is located on the Processor Controller card and performs specialised system functions for both processors. These include serial and

Parallel I/O, floppy disk operations, processor interaction, and program debug facilities.

The remaining memory space is available for the user's RAM. The 64K cards allow separate blocks of 16K physical memory to be assigned, under program control, to any 16K boundary in the address space of either, both, or neither of the processors. This allows each processor to access more than 64K by switching different blocks in and out of the same address space. Each memory module is assigned a card number (Ø to 15) with a hardware switch, allowing up to 64 blocks of 16K to be uniquely identified. It is therefore theoretically possible to use up to 1 megabyte of RAM, even though the 6800 can only address 64K bytes directly.

The memory map of the QASAR breaks down to two parts: hardware determined and I/O system dependent.

The hardware determined section is the address space for both processors from $F000 up, plus the screen output cards, whichever is/are installed. In the case of a standard QASAR system, the screen output cards are Processor 1 unique, although for special applications they can be optioned for Processor 2 instead.

The I/O dependent address space is that from $0000-EFFF inclusive and is the system RAM area - used by QDOS and user programs. How it is initially allocated is determined in the first instance by the firmware installed. If the IOPACK is loaded, then a different allocation is made. All variants assume mapping RAM card Ø is installed. All RAM blocks on that card are enabled to at least one address space. Other RAM cards in the system will remain disabled unless explicitly enabled by user programs.

Figure 1.2 shows the QASAR memory map as configured by the firmware on power-up or RESTART. Note that if the Graphics option is installed, no P1 ram above $8000 is available. If the T.V.T. option is installed instead of the graphics, P1 RAM is available from $8000 to $E000 (See figure 1.3).

For detailed description of use of the Mapping Memory, refer to the Q096 64K RAM card data sheet.

1.2 Firmware

The QASAR ROMs, located on the Processor Controller Card, provide a debug monitor program as well as subroutines for peripheral and floppy disk I/O functions.

The debug monitor (see section 5) is entered by pressing the console interrupt button, which sends an NMI to processor 2. This halts a program running in processor 2 and allows the user to examine and change registers and memory locations, set breakpoints in the program, and restart it from the point where the NMI occurred.

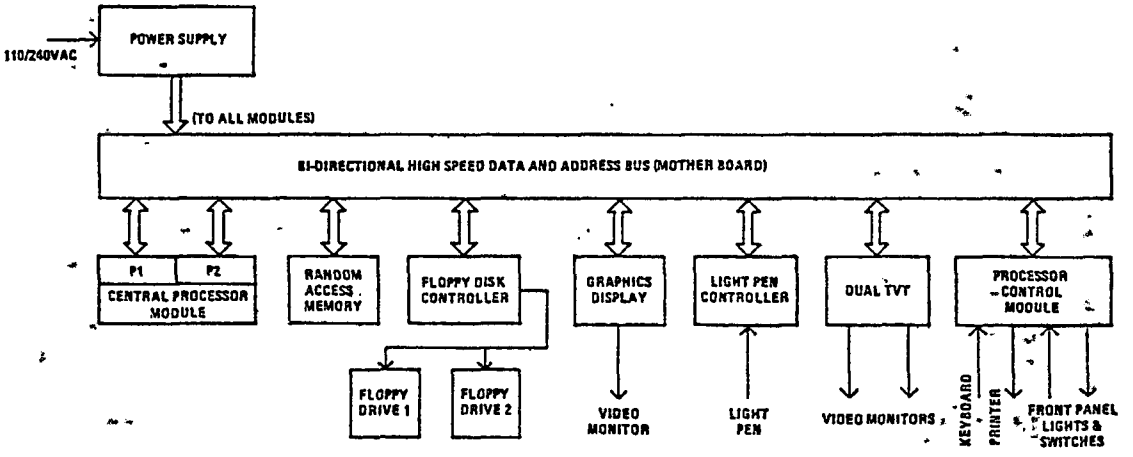


Figure 1.1 QASAR BLOCK DIAGRAM
- Both Graphics and T.V.T. options are shown.

Standard QASAR firmware partitions the system such that P1 is an I/O processor while P2 runs the 'system' and disk I/O. The machine can be simply regarded as a single processor (P2) computer attached to sophisticated terminal and line printer driver hardware whose internal workings remain transparent to the user.

The primary firmware functions supported by Processor 2 are:

-to enable orderly system start-up.
-for keyboard input, and output to console and line printer. These ROM calls communicate with the P1 I/O routines via simple transfer locations in common RAM.
-for floppy disk transfers to and from memory.
-for hexadecimal debugging. A 'monitor' ROM is installed allowing in-memory debugging of user programs.
-to start up (Boot) the disk operating system automatically after Restart.

P1 has firmware to enable it to operate as the I/O processor. The characteristics of the I/O system depend on whether the user has opted for a TVT or graphics screen as the console output device.

The TVT is a memory-mapped device where ASCII character codes are converted by hardware to characters on the screen. The advantage is speed of operation.

The graphics device dumps an area of memory data bit by data bit to the screen. Images, such as those of characters, are written as bit patterns into that memory and appear drawn on the screen. With this option installed, P1 may run a special I/O program (the IOPACK), loaded off disk, which converts the screen into a graphics terminal. While still allowing normal character output, it will interpret commands which allow vector and point plotting, flexible character and cursor modes, protected field I/O, and light-pen processing.

For full details of the IOPACK functions, refer to section 3.

All P1 firmware supports 32-character keyboard queuing. The TVT supports a 2 Kilobyte line printer queue. The graphics system supports approximately 12K of line printer queuing if the IOPACK is loaded.

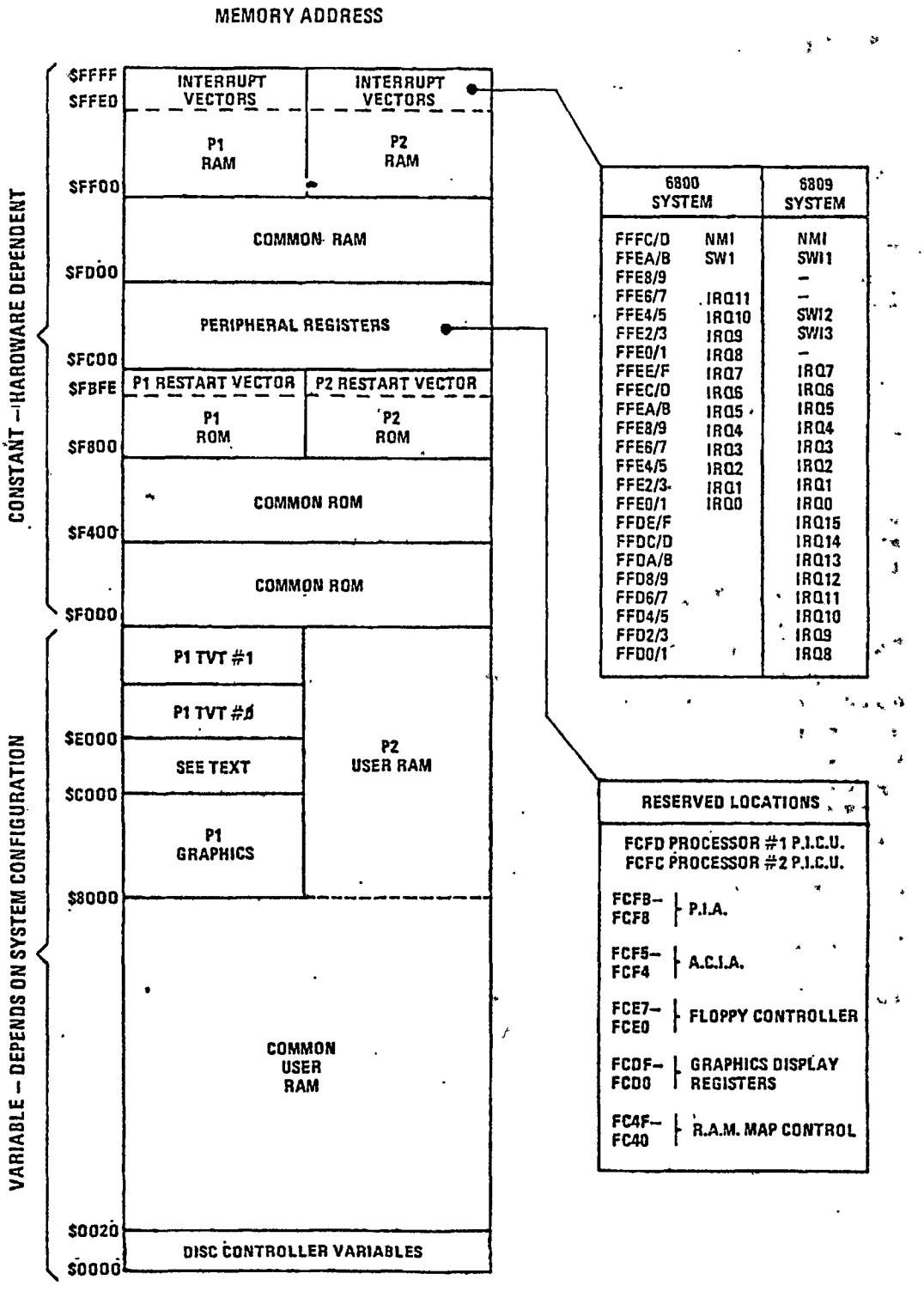


Figure 1.2 QASAR MEMORY MAP - HARDWARE ALLOCATION
- RAM Mapping is configured by firmware on startup.

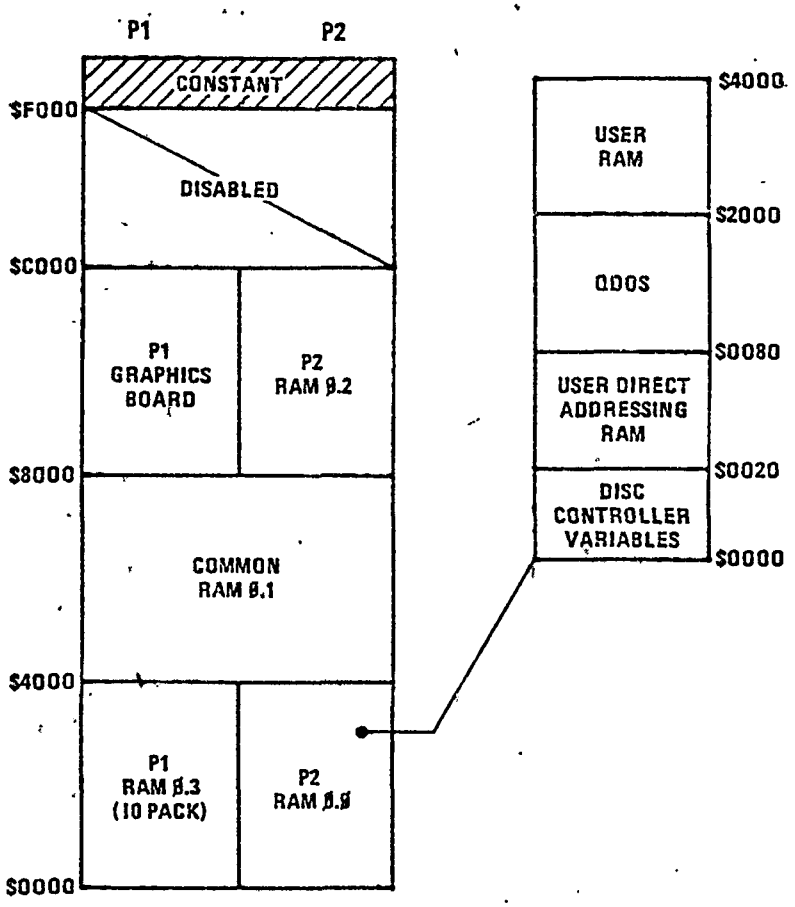


Figure 1.3 QASAR MEMORY MAP WHEN RUNNING GRAPHICS SYSTEM - QDOS and IOPACK are both loaded. The RAM block numbers are in the format: <card.block>

1.3 Dual Processor Operation

In normal operation, user programs are executed by processor 2 while processor 1 executes an I/O loop handling the keyboard, TVT display and printer, and updating the time-of-day clock. Processor 2 invokes these functions by calling routines in common ROM which perform the necessary communication with the other processor. Keyboard and printer data is queued by processor 1, allowing processor 2 to continue with other tasks while the peripherals are kept busy.

A user program may be run in processor 1 by loading it into common memory and storing the start-of-execution address in locations $FE01 (Hi) and $FE02 (Lo). Processor 1's I/O handling loop includes a test of a trigger byte at location $FE00. When this byte is set to non-zero through processor 2, processor 1 saves a return address for the I/O loop and jumps to the address stored in locations $FE01,2.

All keyboard, screen, printer, and time-of-day clock operations will be suspended until processor 1's I/O handling loop is re-entered. A return address for this purpose is stored in locations $FE03 (Hi) and $FE04 (Lo). When the user program in processor 1 jumps to this address, the trigger byte is cleared and normal I/O operation is resumed. The start address at locations $FE01,2 is preserved so that the user program may be re-entered by setting the trigger byte to non-zero again.

The time-of-day clock may be kept accurate without returning to the I/O loop by calling the subroutine EXROUT at location $F803 in processor 1's unique ROM area. Processor 1 must execute this routine at least once every 15ms to ensure that clock "ticks" are not missed.

1.4 Using The 6800/6809 System

The 6809 is a third generation microprocessor. It is easier to program than the 6800 and faster in execution due to its enhanced instruction set, addressing modes, and 16-bit arithmetic capability.

A QASAR computer fitted with the 6800/6809 processor card provides facilities for developing and running programs in either or both processors with no hardware changes or adjustments. The computer is designed primarily as a development system for either 6800 or 6809 programs with the processor configuration under full software control.

Normally the 6809 functions as processor 1 and after restart is waiting in a loop while processor 2 runs the disk operating system and other 6800 programs. The 6800 handles disk and peripheral I/O without queueing, just as a single processor computer would.

The QDOS "LOAD" command can be used to load a 6809 program from floppy disk into any available area of common memory (memory restrictions are outlined in section 2.1). Control can be passed to the 6809 by specifying the "U" option to enter the 6809 monitor, or the "UG" options to start execution of the loaded program.

When a program is loaded with the "U" option, the 6800 is reconfigured to operate as an I/O processor for the 6809. All calls to operating system functions or ROM subroutines from the 6809 program are detected and executed by the 6800. This operation is automatic and of no concern to the user. It provides I/O compatibility between the processors, allowing existing 6800 programs to be run on the 6809 with a minimum number of changes.

Peripheral I/O functions available to 6800 programs through ROM calls to locations $F000-$F02A (see section 5) have been duplicated for the 6809 at locations $F400-$F42A, i.e. the same calls may be used in 6809 programs provided an offset of $400 is added. Floppy disk operations have the same entry points for both processors ($F800-F82C). The line printer routines available to 6800 programs as ROM calls to locations $F7D6-$F7F4 are not available to 6809 programs, however these functions may be invoked through system calls which are identical for both processors. There are no time-of-day clock routines in the 6800/6809 system.

Separate monitor ROMs are provided for the two processors as their machine codes are incompatible. The 6800 monitor may be entered directly by pressing the console interrupt button (processors 1 and 2 must both be interrupted). The 6809 monitor may then be entered by typing the command "M". This switches the 6800 to an I/O handling routine for the 6809. The monitors are almost identical in operation (see section 5).

The 6800 and 6809 processors may be used simultaneously through the trigger mechanism described in section 1.3. The starting address of a memory-resident 6809 program is placed in locations $FE01 (Hi) and $FE02 (Lo) using the 6800 processor, and the trigger byte ($FE00) is set to non-zero to start the 6809 executing. The 6800 continues to operate independently and does not enter the I/O servicing mode for the 6809. Hence the triggered 6809 program cannot perform any peripheral or disk I/O and must use common memory locations for communication with the other processor.

1.5 Interfacing

The Processor Controller card provides two serial outputs and one serial input as well as two 8-bit parallel bi-directional ports. In normal operation the serial input is used by the keyboard and one serial output controls a printer.

Prototyping cards (8"x 8") are available for custom built interfaces and these plug into any of the nine spare slots on the motherboard. All slots are pre-wired with data, address, timing, control and power supply lines. The QASAR power supply provides 5 V at 10 A (15 A option available) and +/- 12 V at 1.0 A (1.5 A option available).

Various special purpose interface cards are available for the QASAR.

1.6 Floppy Disk Operations

All disk operations are handled by DMA through the floppy disk controller module and routines in processor 2's unique ROM. Each DMA cycle is achieved by stretching processor 2 clock phase 1 by one microsecond. This makes DMA completely transparent to the processors, saving the time required to do a Request/Acknowledge cycle that would otherwise be required.

A user program normally accesses the floppy disks by calls to the QDOS operating system which maintains file directories and sector data buffers. QDOS supports single and double sided floppy disks with a maximum capacity of 250K bytes (single-sided) or 500K bytes (double-sided) per disk. Up to 160 files may be stored on each diskette, and files may range in size from 512 bytes up to the maximum capacity of the disk.

1.7 Front Panel Controls

(Refer to Figure 1.4)

Power supply LEDs: These monitor the +5 and +/-12 V power supplies.

Restart switch: This pushbutton switch causes either or both processors to re-enter the ROMs at points specified by the restart vectors in each unique processor ROM (location $FBFE). Normally both processors are restarted. Processor 2 will initialise and run the QDOS operating system on restart if a QDOS system disk is placed in the left hand drive. Power-on restart always resets both processors and all peripheral controllers.

Console Interrupt switch: This pushbutton switch causes a non-maskable interrupt to be issued to either or both processors. Normally only processor 2 is interrupted so that screen and keyboard operation is retained. The NMI enters the debug package (monitor) in common ROM.

On the 6800/6809 QASAR, both processors must be interrupted and the 6800 monitor is entered initially. The message A;hhhh will be displayed, where hhhh is the contents of the program counter at the time the interrupt occurred. The contents of the CPU registers and memory may be examined and/or changed using the Monitor. the command P will cause the CPU to Procede with the user program from the point at which it was aborted.

Processor 1 and 2 Run/Halt switches: These control the run/halt signals to each CPU and are normally set to "Run".

Processor 1 and 2 Wait LEDs: These reflect the "Bus Available" signal from each processor. They light when the respective processor is halted or has encountered a WAIT instruction.

Terminal Speed: This 8-position switch sets the baud rate for all serial data lines and is normally set to 9600.

Printer Mode: This switch provides a manual override of the software controlled "printer start" signal and is used to turn off unwanted printout. It is normally set to "Auto".

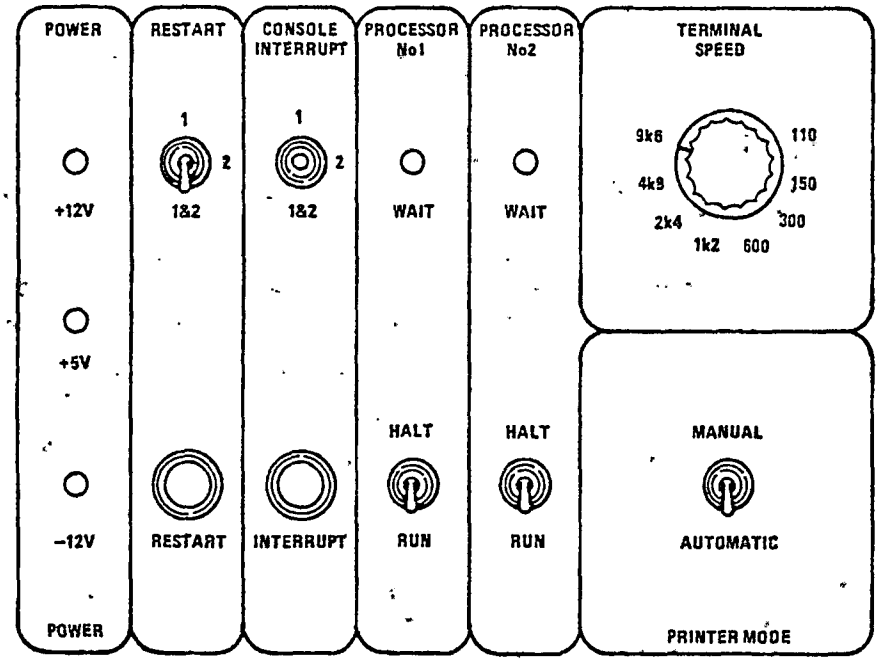


Figure 1.4 QASAR FRONT PANEL CONTROLS

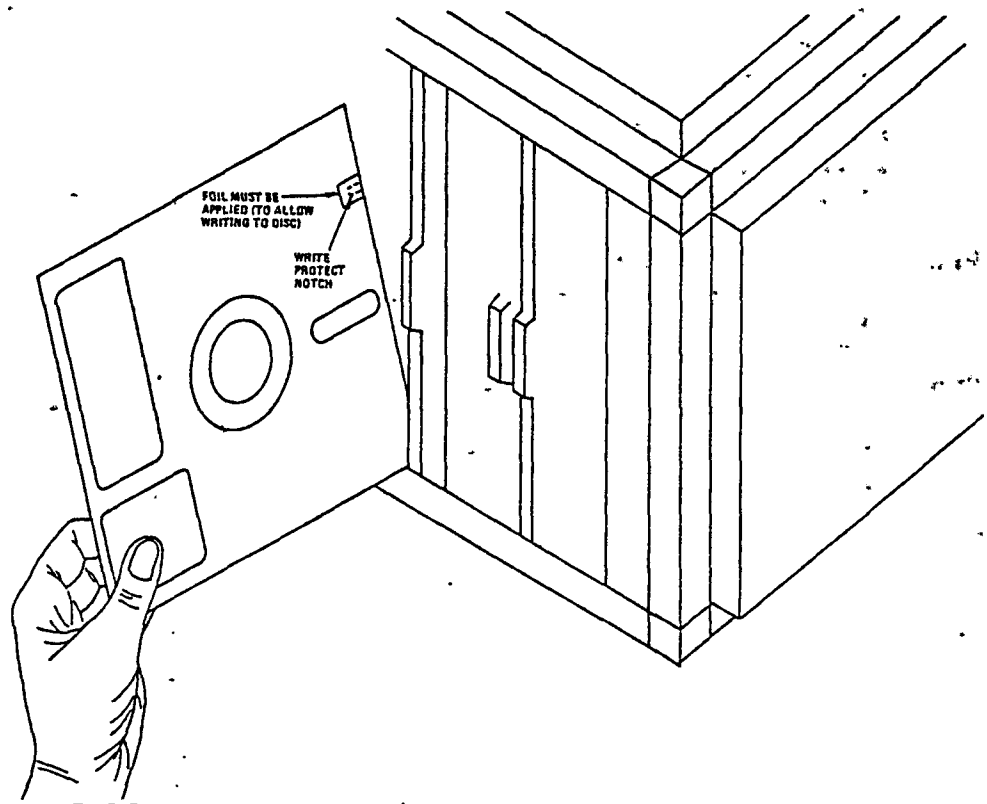


FIGURE 1.5 DISC INSERTION / REMOVAL TECHNIQUE

1.8 System Startup Procedure

a) Connect Video Display, Alphanumeric keyboard, and Line Printer (if used).

b) Ensure that Mains key switch located on rear panel is in the OFF position.

c) Set all front-panel controls as described in Section 1.7 (above). Refer to Figure 1.4.

d) Switch power ON. Note that the key can be removed with the switch in the ON position if desired.

e) The system start-up prompt will be displayed on the screen. This means the system is now ready to boot the disk operating system.

e) Insert a QDOS System Disk in Drive #0 (the left-hand one) as shown in Figure 1.5. Push the disk all the way in until it clicks home. Latch the drive door firmly shut.

f) The QASAR will now load the DOS automatically and display the QDOS sign-on message when it is ready to accept operator commands.

If a fatal disk error occurs during the bootstrap sequence, messages as described in the QDOS manual will be displayed.

** WARNING ** Diskettes should not be left in the drives when power is being switched on or off as there is a risk of data corruption. 1.9 Interrupts

The QASAR microcomputer system is provided with an expandable prioritised interrupt system.

Interrupt prioritisation and automatic vectoring is provided by 8214 Priority Interrupt Control Units located on the Processor Control card. The standard configuration supports 8 levels of interrupt.

As a general rule, the QDOS operating system does not operate with interrupts, so user programs using interrupts should ensure that the interrupt mask bit is SET during system calls. If necessary, interrupts may be used provided the instructions in the QDOS manual are followed.

1.9.1 Using Interrupts with the Dual 6800

Interrupt vector addresses are:

ADDRESS	FUNCTION
FFFE/F	Selects restart vector in unique ROM for particular CPU (at location $BFFE).
FFFC/D	RAM address containing NMI vector.
FFFA/B	RAM address containing SWI vector.
FFF8/9	Unused (reserved for level 12).
FFF6/7	Unused (reserved for level 11).
FFF4/5	Unused (reserved for level 10).
FFF2/3	Unused (reserved for level 9).
FFF0/1	Unused (reserved for level 8).
FFEE/F	IRQ priority level 7 vector (lowest).
FFEC/D	IRQ priority level 6 vector.
FFEA/B	IRQ priority level 5 vector.
FFE8/9	IRQ priority level 4 vector.
FFE6/7	IRQ priority level 3 vector.
FFE4/5	IRQ priority level 2 vector.
FFE2/3	IRQ priority level 1 vector.
FFE0/1	IRQ priority level 0 vector (highest).

These vectors are stored in each processor's unique RAM, located on the Processor Control card. The user's program must initialise these vectors with the address of the interrupt service routine corresponding to each level of interrupt. Levels 0 to 7 are controlled by Priority Interrupt Control Units (PICUs) located on the Processor Control card. The remaining five levels can be provided by a second PICU elsewhere in the system.

The user's program establishes an interrupt priority level by writing a number from 0 to 7 into the PICU Current Status Register ($FCFC for processor 1, $FCFD for processor 2). This number is the ONES COMPLEMENT of the desired mask level, i.e. writing a $07 to $FCFC will enable all levels of interrupt to processor 1, and writing a $03 to $FCFD will enable interrupt levels 0,1,2,3 and 4 to processor 2.

When an interrupt comes in on a level above the highest masked level, the appropriate processor will fetch the IRQ vector corresponding to that level.

The Interrupt Latch is reset by writing a new Current Status to the appropriate PICU. This must take place after an interrupt st that the prioritiser will be re-armed. To properly dismiss an interrupt, the interrupt routine should first reset the interrupting device (e.g. ACIA, PIA), then write the desired mask level to the PICU before performing the RTI.

1.9.2 Using Interrupts with the 6800/6809

Operation of interrupts using the 6809 is performed in much the same way as in the dual 6800 system, with the exception that the memory map for interrupt vectors is expanded to accommodate the fast IRQ and the two extra software interrupt vectors required for the 6809.

The interrupt vector addresses for processor 1 (6809) are:

ADDRESS	FUNCTION
FFFE/F	Selects restart vector in unique ROM for processor 1 (at location $BFFE).
FFFC/D	RAM address containing NMI vector.
FFFA/B	RAM address containing SWI1 vector.
FFF8/9	Unused.
FFF6/7	RAM address containing FIRQ vector.
FFF4/5	RAM address containing SWI2 vector.
FFF2/3	RAM address containing SWI3 vector.
FFF0/1	Unused.
FFEE/F	IRQ priority level 7 vector.
FFEC/D	IRQ priority level 6 vector.
FFEA/B	IRQ priority level 5 vector.
FFE8/9	IRQ priority level 4 vector.
FFE6/7	IRQ priority level 3 vector.
FFE4/5	IRQ priority level 2 vector.
FFE2/3	IRQ priority level 1 vector.
FFE0/1	IRQ priority level 0 vector (highest).
FFDE/F	IRQ priority level 15 vector (lowest).
FFDC/D	IRQ priority level 14 vector.
FFDA/B	IRQ priority level 13 vector.
FFD8/9	IRQ priority level 12 vector.
FFD6/7	IRQ priority level 11 vector.
FFD4/5	IRQ priority level 10 vector.
FFD2/3	IRQ priority level 9 vector.
FFD0/1	IRQ priority level 8 vector.

The vector addresses for processor 2 (6800) remain the same as for the dual 6800 system (above).

-SECTION 2-

2.0 THE QDOS OPERATING SYSTEM

QDOS is an interactive operating system that obtains commands from the system console. These commands are used to move data on the floppy disks, to process data, or to activate user-written programs.

On every QDOS diskette there are nine files which comprise the operating system. These files contain the resident operating system, a series of overlays to reduce the main memory requirements of the system, and standard error messages. There is also a file directory containing the names, locations and format descriptions of all files on the diskette. A Retrieval Information Block (RIB) is associated with each file and specifies which disk locations are used by the file (these are not necessarily contiguous). Executable binary files also have a starting load address and a starting execution address contained in the RIB.

QDOS files are identified by a name followed by a two letter suffix which defines the file type, and an optional disk drive number. Allowable file types include program source or ASCII data, loadable binary object files, and chain or procedure files containing a sequence of QDOS commands. The type of file is specified in the attribute field of the directory entry. Certain QDOS functions assume a default suffix for a particular file type (e.g. .SA, ASCII file) but any suffix may be used provided the full name is entered.

All QDOS commands are executable binary files with the suffix .CM. When a command file name is typed in (the suffix is optional), the command interpreter searches the directory and loads and executes the specified command. QDOS provides facilities for editing and assembling programs so that new processes can be written and invoked as operating system commands.

Commonly used tasks such as I/O operations, string handling, and directory and file accessing are performed by QDOS through system function calls (SCALLs) in the user's program. An SCALL is a software interrupt instruction followed by a byte containing the number of the system function to be executed. QDOS provides over sixty of these and additional SCALLs can be defined by the user. A unified I/O procedure is provided through SCALLs, enabling device independent programs to be written.

Section 2: THE QDOS OPERATING SYSTEM

2.1 QDOS Memory Restrictions

QDOS requires at least 16K of contiguous memory to run. The resident part of the operating occupies memory locations $0100 to $1FFF (8K) and this area must be preserved by user programs if they use any QDOS functions. Locations $0000 to $001F are reserved for the variables of the floppy disk controller. These locations cannot be initialised by a program loading from diskette. In addition, if a program requires the use of the diskette functions (either directly through the floppy disk controller firmware or through the QDOS functions), then these locations cannot be used by the program for storage.

Command-interpreter-loadable programs must load above location $1FFF. They can use the direct addressing area (below $0100) for variable storage; however this area cannot be initialised while the program is being loaded into memory. Programs that do not make use of QDOS system functions can be loaded anywhere into memory above location $001F using the "LOAD" command and "V" option, which allows resident QDOS to be overwritten. If such programs do not use the floppy disk controller entry points (section 3.3), the direct addressing area below location $0020 can be used, but only after the program is resident in memory.

2.2 Writing and Running a Program with QDOS

The QASAR computer supports all the software tools necessary for a complete program development system. The QDOS operating system provides the basic man-machine interface as well as file management and process control. A Text Editor allows source programs to be entered, modified, and stored on diskette in ASCII format. The Macro Assembler converts program statements into machine code which is stored on diskette and can be loaded and run by the operating system. A Linking Loader permits large programs to be split into a number of separate modules which can be developed and tested individually. The QASAR ROMs provide some debug facilities including breakpoint controls and instructions to examine and change memory locations.

Each of these software products is described in detail in a separate manual but the following simple example illustrates the general procedure for writing and running a program on the QASAR.

Stage 1- Editing

In this example a program is typed in on the console keyboard and stored in a diskette file named TEST. The program statements are 6800 assembler mnemonics. Two SCALLs are used to invoke operating system functions.

```
=EDIT TEST

@I NAM TEST
 OPT NOP SUPPRESS PAGE HEADINGS
 ORG $2000 STARTING LOAD ADDRESS
START LDS #STACK LOAD STACK POINTER
 LDX #STRING POINT TO STRING TO PRINT
 SWI
 FCB $0A SCALL TO DISPLAY STRING
 SWI
 FCB $1A SCALL TO RETURN TO QDOS
 SPC 1
STRING FCC / YOU'RE WELCOME /
 FCB $0D STRING TERMINATOR
 RMB 64 RESERVE A STACK OF 64 BYTES
STACK EQU *
 END START START EXECUTION ADDRESS
$$
@E$$
```

The source program will now be saved on the drive 0 disk as the file TEST.SA.

Stage 2- Assembling

In this example the A option in the command line causes the assembler to produce an absolute binary object file which is given the name THANKS.CM and saved on the same drive as the source program (Ø by default). The program listing has been sent to the console screen but can be directed to the line printer or a disk file.

```
=RASM TEST;AO=THANKS.CM,L=#CN

00001                              NAM     TEST
00002                              OPT     NOP       SUPPRESS PAGE
00003A 2000                        ORG     $2000     STARTING LOAD
00004A 2000 8E 205B  A START       LDS     #STACK    LOAD STACK
00005A 2003 CE 200A  A             LDX     #STRING   POINT TO STRING

00006A 2006 3F                     SWI
00007A 2007    ØA    A             FCB     $ØA       SCALL TO
00008A 2008 3F                     SWI
00009A 2009    1A    A             FCB     $1A       SCALL TO

00011A 200A    20    A STRING    FCC     / YOU'RE WELCOME /
00012A 201A    ØD    A           FCB     $ØD       STRING
00013A 201B    0040  A           RMB     64        RESERVE A STACK

00014          205B  A STACK     EQU     *
00015                              END     START     START EXECUTION

TOTAL ERRORS ØØØØ--ØØØØØ
```

Stage 3- Running the program.

Program execution is accomplished by simply typing in the object-code file name. The QDOS command interpreter is expecting the name of a file containing a memory image with valid load and start addresses. If these conditions are satisfied, the file is loaded from diskette into memory and control is given to the start-of-execution address. The program terminates by returning control to the operating system with an SCALL, and QDOS prompts the operator for another command.

```
=THANKS
 YOU'RE WELCOME
=
```

2.3 Program Debugging

The QASAR computer provides several aids for program debugging. The monitor ROM program (see section 4) may be entered from a user program running in processor 2 by executing a jump to the monitor starting address or by interrupting the processor by pressing the console interrupt button. Processor 1 should not be interrupted in a dual 6800 system as it handles the screen and keyboard I/O for the monitor. The 6800/6809 system has separate monitor programs for each processor with a facility for switching between monitors. Both processors should be interrupted and the 6800 monitor is entered initially.

Another useful aid to program debugging is the screen dump. User programs running in processor 1 may write directly to the screen VRAM in the dual 6800 system. The QDOS function SDUMP may be used by processor 2 in the dual 6800 system or either processor in the 6800/6809 system.

SDUMP reserves an area of the TVT screen for real-time continuous display of a selected group of memory locations. This data is maintained by processor 1 in the dual 6800 system or processor 2 in the 6800/6809 system. SDUMP allows optional display of 6-character labels along with the data, to avoid confusion when large amounts of data are being displayed.

The SDUMP can be directed to either of the two TVT displays. If it is directed to the console screen ($D000), the SDUMP information appears at the top of the screen, occupying as many lines as required, and the remaining lower section of the screen operates as a normal scrolled display, except that the number of lines is reduced accordingly.

3.Ø QASAR GRAPHICS / LIGHT-PEN SYSTEM

If the QASAR Graphics hardware is installed, a wide range of powerful I/O functions can be easily accessed via the IOPACK software interface. By using simple ESCAPE sequences, complex plotting and light-pen functions are easily achieved. When the IOPACK is loaded, P1 begins executing a sophisticated set of service routines. Features of the IOPACK include:

- Better screen density - more characters are displayable.
- Graphics terminal which executes simple ASCII commands.
- Easy to use light-pen interface.
- Large (approx. 12K) line printer queue.
- Operation transparent to the user.
- Interfaces to any programs which use the system console.

3.Ø.1 System requirements

A QASAR computer is required with the following cards installed:

- At least one mapping 64K RAM card.
- P1 only Graphics board and associated 16K RAM card.
- Light-pen board (optional).

3.1 Loading the IOPACK

Whenever the IOPACK is loaded, it forces P1 to execute its ROM. The system memory map is changed to accomodate the IOPACK in mapping RAM block Ø3. That block is then mapped down to address $ØØØØ in P1 unique address space. The top of user memory becomes $BFFF and $4ØØØ-7FFF becomes the only P1,P2 common RAM available to the user. Finally P1 is taken out of its ROM and begins executing the IOPACK service loop. P1's stack is always assumed to be in the P1 RAM at $FFØØ upwards.

3.1.1 Loading by command

The IOPACK resides on disk as a command, called QIOPACK and may be invoked at any time. The keyboard queue will not be lost. Any line printer queue will be lost. The top of user RAM will be set to $BFFF. Otherwise, the action is transparent to P2.

3.1.2 Auto-load by QDOS

QDOS as supplied with QASAR graphics systems will automatically load QIOPACK, if found on the system disk, when the system is restarted (booted). The action may be suppressed if the user requires memory rather than enhanced I/O, by giving QIOPACK another name. Then QDOS will not find it at boot time. The user can then use his new name to load the IOPACK as a command if desired.

3.1.3 When not using QDOS

If the user wishes to use the IOPACK with a different disk operating system (such as UCSD Pascal) then it is accomplished by loading the IOPACK first under QDOS. Remove the QDOS system disk and restart P2 ONLY. P1 will continue executing the IOPACK while P2 will be ready to boot the other operating system.

3.2 Keyboard

The keyboard behaviour remains unchanged. The transfer location (KFERPT at $FE2A) when non-zero, has the next pending keyboard character ready for P2 to pick up.The Status Flag (STFLAG at $FE1A) is set immediately to indicate that a "break" or special control code has been typed at the keyboard. It is cleared when any other character is typed.

3.3 Line Printer

A large printer queue is maintained. Spaces are compressed when queued. The printer is checked for status. Automatic motor control is maintained for some printers such as the Teletype Model 40. These will be powered down about 4 seconds after the last character is sent to them.

3.4 Video Screen

The screen responds to:

Characters - which are printed.
Escape sequences - which are executed.
Control codes - such as CR, LF and FF.

The IOPACK maintains a graphics pen, and two cursors. All may be used independently.

3.5 Character output

In general, characters sent to the screen behave as expected and appear at the current cursor position. The only times that they are not printed are when they are part of an escape sequence, or character output has been suppressed.

Characters are 5 x 8 dot matrices with the bottom line reserved for the underline cursor. They are justified to the bottom left corner of a character cell which is initialised to be 6 dots wide by 8 dots high.

Screen output normally goes to the bottom line of the display, which scrolls upward.

3.6 Escape sequences

Escape sequences are sent to the IOPACK system via the normal console character output routines, that is system calls or ROM calls from Assembler programs, or PRINT statements in BASIC.

Whenever an ESC character ($1B) is sent to the screen, then all characters up to and including the next carriage return and line feed, are interpreted as commands to the IOPACK. Multiple commands may be sent on the one line: with an ESC preceding each. Invalid commands and parameters are ignored.

Commands take the form of a two letter code followed by a variable number of signed decimal integers. Plus signs are not required. Delimiters are required only where there is ambiguity. The minus sign of a second parameter is a good enough delimiter from the first. Valid delimiters are spaces and commas. Spaces (if any) may separate the command from its first parameter. Thereafter spaces and commas are allowed between parameters. All other characters will terminate parsing prematurely.

For example, the following are all equivalent:

```
PA 50,-80          PA 50-80
PA50-80            PA    50        -80
```

3.7 Pen versus cursor

It will quickly become apparent to the user of escape sequences that the screen actually has two independent modes of operation: graphics and character. Graphics output is drawn by a 'pen'. This should not be confused with the light-pen. Character output occurs at a 'cursor'. These terms will be used throughout the following sections.

3.8 Commands by group

3.9 Screen colour

3.9.1 BC n Set background colour
3.9.2 IN Invert screen image

The QASAR graphics display is green-on-black, so the word "colour" in this context is actually reference to the bit pattern written to the screen. The result of differnt "colours" is therefore a shading effect.

The screen background colour is defined by an 8-bit binary number called the background mask. The BC command is used to assign a value (bit pattern) to the mask. Clear bits are black. Set bits are green. When a Formfeed ($0C) is sent, then the mask is written out to every byte of the graphics RAM, which is in turn displayed on the screen. The action is that of clearing the screen and initialising it to a "colour". If the background mask is $00 then the screen will be all black. If the colour set is -1 (mask = $FF), then the screen will be all green. Intermediate values result in stippling across the screen.

Most marks made by the cursor and pen will be written to contrast with the background colour. Characters will be green on a black background until the background mask is inverted ($00 --> $FF) when they will become black on a green background. Similarly, if the pen colour is non-zero, it will always make a visible mark. The pen colour mask is exclusive OR'ed with the background mask before being written to the screen.

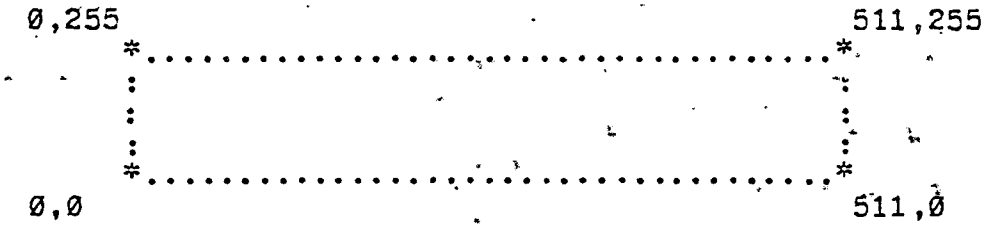
The IN command inverts every every bit on the screen as well as the background mask.

Defaults: n = 0

BC - sets the background to black.

3.10 Graphics group

The pen is one dot high and wide and may be moved around the screen at will. The screen is 256 dot positions high by 512 wide. All co-ordinates are specified in X,Y form where 0,0 is the bottom left corner of the screen.



Co-ordinates which pass screen boundaries will wrap around.

The pen may be dragged across an area of screen (PLOTTING) leaving a mark behind it if the pen colour is non-zero. Alternately the pen may be lifted and MOVED to a new position. Finally, the pen may be dabbed quickly against the screen to plot a single dot.

Provision is made to drive the pen using absolute or relative co-ordinates. An offset register is also maintained and is added, within the IOPACK, to every absolute parameter received. Take the case of the move commands:

```
MA x,y - Move absolute      newpos := offset plus (x,y)
MR x,y - Move relative      newpos := oldpos plus (x,y)
```

3.10.1 Pen colour

3.10.2 PC n - Set pen colour.

Pen mask is loaded with the 8-bit 2's complement value of n. The pen mask is exclusive OR'ed with the background mask before being written to the screen. n=-1 sets all 8-bits ($FF) and so what is written will always be the opposite of the background colour, that is it will contrast. n=0 will draw lines the same colour as the background - good for erasing selectively. Intermediate values will give 8-dot repeating patterning along the horizontals.

Default: n=-1

PC - switches pen 'on', will always make a mark.

3.10.3 Offset register

3.10.4 OF x,y - Set offset to x,y.
3.10.5 OR x,y - Add x,y to offset.

The offset may be loaded with a specified value or changed by a specified amount. The new value of the offset is then added to all absolute co-ordinates specified by subsequent graphics commands.

Defaults: x,y = 0,0

OF - Clear offset register
OR - No effect.

3.10.6 Move pen

3.10.7 MA x,y - Move pen absolute.
3.10.8 MR x,y - Move pen relative.

Move pen to a new position, leave no marks. Equivalent to lifting the pen before moving.

Defaults: x,y = 0,0

MA - Move to current value of offset register.
MR - No action.

3.10.9 Vector plot

3.10.10 PA x,y - Plot absolute.
3.10.11 PR x,y - Plot relative.

Plot a vector from current position to new position specified. A vector, as straight as possible, is drawn. The action is not commutative. A vector between the same two points but drawn in the opposite direct will not exactly overwrite the original.

Defaults: x,y = Ø,Ø

PA - Draw a vector to current value of offset.
PR - Plot a dot at current pen position.

3.10.12 Dot (point) plotting

3.10.13 DA x,y - Plot a dot absolute.
3.10.14 DR x,y - Plot a dot relative.

Plot a single dot at the position specified. Then pen is moved before plotting.

Defaults: x,y = Ø,Ø

DA - Plot a dot at current value of offset register.
DR - Plot a dot at current pen position.

3.11 Character group

Characters are 5 x 8 dot matrices. The bottom line of the matrix is reserved for the underline cursor. Each character is printed at the current cursor position in a character cell whose size may be varied. The character patterm is butted against the bottom left corner of the character cell. Default character cell size is 6 dots wide by 8 dots high.

All cursor co-ordinates are specified by line number and character position along the line. These co-ordinates will refer to different locations on the screen if the character cell size is changed. Line 1, character 1 is the character cell butted up to the top left corner of the screen and is the reference point.

The bottom left character cell on the screen may not always sit on the physical bottom line, and yet that is where screen output is often desired to be sent. Therefore, line Ø, character Ø is an artificial co-ordinate to reference that location.

The screen may be split into a scrolling area, relatively loosely controlled, extending up from the bottom and a fixed non-scrolling area extending from the top down. The cursor address reference is at the top for this reason. I/O fields would usually be defined in this area.

Character output may be switched off to suppress keyboard echo. The underline cursor can also be suppressed.

Finally there are two banks of cursor control registers which contain the following information:

Cursor position
Cursor mode
Character cell size
Protected field info. (if any)

These may be swapped at any time by sending a SYN character (cntl V or $16). Two independent cursors may be maintained. The swapping is transparent to IOPACK commands which will reference whichever bank is current.

A useful application is the setting aside of a special area for error messages to go. The messages need only be preceded by and finish with a SYN character. Initially, the alternate bank is set to a pseudo-protected field on the bottom line of the picture, with the cursor suppressed.

3.11.1 Cursor Mode

3.11.2 CU n - Set cursor mode.

This is a cursor control command. Subsequent cursor behaviour is:

n=-1 : no character output or cursor.
n=Ø : characters will print, cursor suppressed.
n=1 : both characters and cursor on. (normal operation)

Default: n=Ø

CU - cursor off, characters will still print.

3.11.3 Cursor movement

3.11.4 MC l,c - Move cursor to line,column.

3.11.5 CP - Move cursor to pen position.

MC moves the cursor to the line and character position specified. The exact physical screen location depends on the current character cell size.

CP links the cursor to the graphics pen without reference to line and character position boundaries. This facilitates the labelling of graphs. The next character printed will rest with its bottom left dot at the graph pen position at the time this command was received.

Defaults: l,c = Ø,Ø

MC - Resets the cursor to butt into the bottom left corner of the screen.

3.12 Scrolling area

3.12.1 TS 1 - Set top of scroll

The scrolling area is set up to and including the line specified by 1. Thus this command is used to split (and restore) the screen.

When split, the scrolling area is software scrolled and, if a significant size, may scroll quite slowly. It is best to keep the scrolling area to the minimum required.

Whole screen scrolling is accomplished by hardware and is therefore fast.

Default:

TS - Sets the whole screen to scroll.

3.13 Protected Fields

3.13.1 PF line,char,len - Define a field.
3.13.2 GF field - Move cursor to a field.

Fields are defined by position and length. The cursor may be moved into them and special limits on its movement apply. Character output then proceeds as normal. These areas may be used for I/O or be labelled and referenced by the light-pen.

PF enters the parameters supplied into a field table within the IOPACK. 256 such entries are allowed. The field table is sorted on line, character position and order of entry (where there are two more entries for the same position).

Field numbering then is from the top down and left to right across the screen.

```
< F 1 >    < F 2 >

  < F 3 >
< F 4 >
```

Field number 1 is the first entry in the field table.

The information remains until a Formfeed is received by the IOPACK. This not only clears the screen but purges the field table.

Bad parameters in a PF command result in no entry being made to the table. PF's are ignored if the table is full.

GF moves the cursor to the first character position of the

specified field. From then on, a new set of rules apply to cursor movement. The only way to leave the field is via another GF command, an MC command or by the SYN control code. A second SYN will return the cursor to the field it came from with all the original limits on movement.

Should there be no entry in the field table for the field number specified, then the cursor is not affected in any way.

Defaults : all parameters = Ø

PF - No action, bad parameters.
GF - No action, invalid field number.

3.14 Light pen group

Light pen commands are requests to the IOPACK to return the current position and/or status of the light-pen. One or two decimal numbers are sent back as though typed by a phantom hand on the console keyboard. In BASIC, for example, the user prints a light-pen command to the screen and then immediately should execute an INPUT statement for the expected number of numeric variables. The phantom hand has the ability to jump the keyboard queue and stay ahead of any waiting keyboard characters.

Care should be taken to match each light-pen command sent with an input request otherwise the IOPACK responses will be nested in an unexpected fashion!!

3.14.1 Light-pen commands

3.14.2 LG m - Return position in form x,y.

3.14.3 LC m - Return line,char. position

3.14.4 LF m - Return a field number.

m is a mode control parameter. Currently, modes are:

m=Ø : don't wait for hit. Return -1's if no hit.
m=1 : wait for hit.

Field numbers returned by LF after a hit may be Ø or a valid field number. Ø (an invalid field number) indicates that a hit occurred but outside any defined fields.

Defaults: m=Ø.

LG,LC,LF - Return co-ordinates if any, don't wait for hit.

3.15 Defaults

Defaults are applied when parameters are missing in a command. Thus:

MC	is interpreted as	MC Ø,Ø
PA	as	PA Ø,Ø
CS	as	CS 6,8

Most defaults are Ø except where noted. Generally missing arguments (i.e. the command alone) are used in a 'reset' sense. For example, <ESC>MC<CR> will restore the cursor to the bottom left corner. <ESC>TS<CR> will restore whole page scrolling. Some arguments (e.g. 'colour' registers) are processed to 8 bits. Unused high bits are ignored.

3.16 Screen Control Codes

The screen will respond to:

CR -($ØD,<return>) Return to beginning of line (or field).

LF -($ØA,<cntl J>) Ignored in fields, else three cases:

1. Above top of scroll - ignored.
2. Below top of scroll - Move down 1 char.
3. At bottom of screen - Case 2. plus scroll.

BS -($Ø8,<cntl H>) Non-destructive move left. Limit at beginning of line (or field).

HT -($Ø9,<cntl I>) Non-destructive move right. Limit at end of line (or field).

VT -($ØB,<cntl K>) Move up one character. Limit at top of screen.

FF -($ØC,<cntl 1>) Clear screen (overwrite with specified background colour). Purge field table and reset cursor position to bottom left corner.

ESC -($1B,<esc>) Interpret subsequent characters as graphics commands up to and including <cr> and optional <lf>.

SYN -($1B,<cntl V>) Swap current cursor control registers with alternate set. Used to guarantee display of asynchronous error messages (e.g. PRINTER NOT READY). such messages should start and finish with SYN (CNTL V or $16).

N.B. If the cursor is moved to a protected field (by GF command) then all control codes are ignored except for CR,SYN and BS. ESC is still processed or there would be no escape !!

3.17 Graphics Command Summary

```
BC n  - set background
IN    - invert screen.

PC n  - set pen colour.
OF x,y - set offset to x,y.
OR x,y - change offset by x,y.
MA x,y - move absolute.
MR x,y - move relative.
PA x,y - plot absolute.
PR x,y - plot relative.
DA x,y - dot absolute.
DR x,y - dot relative.

CS x,y- set character cell size.
CU m  - set cursor mode.
MC l,c- move cursor to line(l), column (c).
CP    - move cursor to pen.

PF l,c,n- Protect field of length n at l,c
GF n    - move cursor into field n.

LG m    - light-pen : return graphics co-ordinates.
LC m    -      "              cursor co-ordinates.
LF m    -      "              field number.
```

3.18 Examples of Use of Graphics

3.19 QASAR Compiler Basic

This sample program illustrates the ease with which the graphics screen may be driven and the light-pen interfaced.

```
DIM ESC$/:1B/,FF$/:0C/
DIM X,Y,LINE,CHAR,FIELD

PRINT ESC$;'BC-1' \! SET SCREEN TO WHITE
PRINT FF$         \! AND CLEAR IT

! NOW SET UP A COMMAND FIELD ON THE SCREEN
PRINT ESC$;'PF';31;10;4
PRINT ESC$;'GF1'              \! MOVE THE CURSOR TO IT
PRINT 'QUIT'                  \! AND LABEL IT
! RESET THE CURSOR POSITION AND TURN OFF CHARACTERS.
PRINT ESC$;'MC';ESC$;'CU-1'

20! LOOP FOR GETTING AND PROCESSING LIGHTPEN HITS
GOSUB 100 \! GET A HIT
IF FIELD=1 THEN STOP  \! IT WAS A HIT IN THE 'QUIT' FIELD

   PROCESS X,Y HERE
        .
        .

GOTO 20

100! GET A LIGHT PEN HIT
PRINT ESC$;'LG1'      \! MODE 1 - WAIT FOR HIT
INPUT X,Y             \! GET BACK CO-ORDINATES
PRINT ESC$;'LF'       \! QUICKLY POLL IN FIELD MODE
INPUT FIELD           \! AND GET BACK ANY FIELD NO
RETURN
```

-SECTION 4-

4.0 QASAR ROM SUBROUTINES

The QASAR ROMs contain the Debug Monitor and routines for peripheral I/O and floppy disk operations. Table 4-1 lists the available routines. Except as stated in the following descriptions, all of these are subroutines and end with an RTS instruction. They may be called from user programs by setting up the appropriate registers or RAM locations for parameter passing and executing a JSR or JMP to the required entry point.

TABLE 4-1. QASAR ROM routines

NAME	ADDRESS	FUNCTION
		COMMON MONITOR ROM AT $F000
STARV	F000	monitor start and entry point
STENV	F003	request input of start and end addresses
CONHV	F006	convert an ASCII hex digit to 4-bit binary
CHXLV	F009	convert MS 4-bits to ASCII hex
CHXRV	F00C	convert LS 4-bits to ASCII hex
INADV	F00F	input 16-bits as up to 4 hex digits
IPCNV	F015	input a character, no parity, with echo
OPCHV	F018	display a character on console screen
OP2HV	F01B	display one byte as 2 hex chars, <spc>
OP4HV	F01E	display two bytes as 4 hex chars, <spc>
PCLFV	F021	display <lf>, <cr>, (nulls as required)
PCLSV	F024	display <lf>, <cr>, (nulls), string @X
PRNSV	F027	display string @X
PSPCV	F02A	display <spc>
PRNUL	F02C	display ASCII null
OPBYT	F02D	display a character without nulls
		COMMON I/O ROM AT $F400
VBOOT	F7D6	reset entire I/O, display boot message
LIST	F7DC	output one character to line printer
LINS	F7DF	output string @X to line printer
LINSC	F7E2	output <cr><lf>, string @X to line printer
VIORST	F7E5	reset the entire peripheral I/O system
VRESET	F7F4	clear console screen, reset current pos
		PROCESSOR 1 UNIQUE START-UP ROM AT $F800
EXREST	F800	reset time-of-day clock
EXROUT	F803	update time-of-day clock
		PROCESSOR 2 UNIQUE DISK ROM AT $F800
OSLOAD	F800	boot load QDOS from drive 0
FDINIT	F803	initialise disk controller electronics
READSC	F806	read full last sector

READPS	F80A	read (specified) partial last sector
RDCRC	F80D	read disk and check valid CRCs only
RWTEST	F810	write test and read with CRC verify
RESTOR	F813	restore head to track 0 and lift
SEEK	F816	seek to specified track
WRTEST	F819	write test (no read back check)
WRDDAM	F81C	write "deleted data" marks
WRVERF	F81F	write sectors, read back and check CRCs
WRITSC	F822	write sectors only (no verify)
CHKERR	F825	error check and error message handler
PRNTER	F82C	error code printout handler

4.1 Peripheral I/O Functions

These routines may be called by processor 2 only. They do not access the peripheral devices directly but move data in and out of queues which are maintained by processor 1.

NAME	ADDRESS	FUNCTION
STARV	F000	This entry point initialises the debug package and peripherals from a restart or power up condition. Control is not returned to the calling program, but is given to the debug package command input routine. This call should be made with a JMP.
STENV	F003	This entry point requests input of Beginning and Ending Addresses in hexadecimal. If the ending address is not larger than the beginning address, the operator prompt is repeated. The result is placed in locations $FFD9 (16-bit Beginning Address) and $FFDB (16-bit Ending Address). The contents of the A, B, and X registers are destroyed by this call.
CONHV	F006	This entry point converts a hexadecimal character in acc A to a 4-bit binary number and stores the result in acc A. The hi-order 4 bits are cleared and the N condition code is cleared to flag success. If an invalid character is supplied as input (i.e. not "0"-"9" or "A"-"F") then acc A is not changed and the N condition code is set to 1 to flag failure. The B and X registers are preserved.
CHXLV	F009	This entry point converts the most significant 4 bits of acc A to an ASCII coded hexadecimal digit character and stores it in acc A. The B and X registers are preserved.

CHXRV F00C This entry point converts the least significant 4 bits of acc A to an ASCII coded hexadecimal digit character and stores it in acc A. The B and X registers are preserved.

INADV F00F This entry point inputs up to 4 hexadecimal characters from the console keyboard queue and converts them to a 16-bit binary address. The most significant 8 bits will be stored into the memory location specified by the index register. The least significant 8 bits will be stored into the next higher memory location. The subroutine returns to the calling program when an invalid character is entered. The contents of acc A and acc B are destroyed. The index register is not changed.

IPCNV F015 This entry point removes one character from the console keyboard queue, clears the hi-order (parity) bit, and stores the character in acc A. There is a "NO ECHO" flag (AECHO) at $FFC9. It must be set to non-zero before each call to INCHV for each character that is not to be echoed to the console screen. INCHV returns with AECHO clear and the B and X registers are preserved.

OPCHV F018 This entry point outputs one character contained in acc A and the required number of nulls to the console screen queue. The contents of acc A, acc B and the index register are preserved.

OP2HV F01B This entry point converts one 8-bit binary byte at the address specified by the index register to two hexadecimal characters and outputs them followed by a space character to the console screen queue. On exit, acc A contains the last character output (a space), the index register is incremented by one, and the B accumulator is unchanged.

OP4HV F01E This entry point converts two consecutive 8-bit binary bytes starting at the address specified by the index register to four hexadecimal characters and outputs them followed by a space character to the console screen queue. On exit, acc A contains the last character output (a space), the index register is incremented by two, and the B accumulator is unchanged.

PCLFV	FØ21	This entry point outputs a carriage return, a line feed, and a null character to the console screen queue. On exit, acc A contains a null character (Ø). The B and X registers are preserved.
PCLSV	FØ24	This entry point outputs a carriage return, a line feed, and a user specified string of data characters to the console screen queue. The character string must start at the address contained in the index register and end with an EOT character ($Ø4). On exit the index register contains the address of the EOT character, acc A contains the EOT character, and acc B is unchanged.
PRNSV	FØ27	This entry point outputs a user specified string of characters to the console screen queue. The character string must start at the address contained in the index register and end with an EOT character ($Ø4). On exit the index register contains the address of the EOT character, acc A contains the EOT character, and acc B is unchanged.
PSPCV	FØ2A	This entry point outputs a space character to the console screen queue. On exit, acc A contains a space character. The B and X registers are preserved.
PRNUL	FØ2C	This entry point outputs a null character to the console screen queue. On exit, acc A contains a null character (Ø). The B and X registers are preserved.
OPBYT	FØ2D	This entry point outputs the character contained in acc A to the console screen queue with no added nulls. The contents of acc A, acc B and the index register are preserved.
VBOOT	F7D6	This entry point resets the entire peripheral I/O system and prints the boot message ("LOAD SYSTEM DISK IN DRIVE Ø"). Control is returned to the calling program.

LIST	F7DC	This entry point sends the contents of the A accumulator to the line printer. If the "paper empty" or "printer not selected" status condition is detected, the message "PRINTER NOT READY" will be output to the console screen and the routine will wait in a loop until the condition is rectified and the character can be sent. The "PRINTER NOT READY" message will then be cleared from the screen before returning to the calling program.
LINS	F7DF	This entry point sends a character string to the line printer. The string is pointed to by the X register and must be terminated with an EOT ($04). If a printer error is detected by LINS it will display the "PRINTER NOT READY" message on the console screen and wait in a loop until aborted or until the error is corrected. The "PRINTER NOT READY" message will be cleared from the screen before returning to the calling program.
LINSC	F7E2	This entry point performs the same function as LINS with the exception that prior to printing the string, a carriage return and a line feed are sent to the printer.
VIORST	F7E5	This entry point resets the entire peripheral I/O system and returns to the calling program.
VRESET	F7F4	This entry point clears the console video screen and resets the cursor to the bottom left hand corner.

4.2 PROCESSOR 1 SPECIAL ROUTINES

In normal operation the P1-unique ROM is executed periodically to maintain certain timers and service the I/O functions. For special applications the following functions may be exploited by the user program:

4.2.1 Time-of-day clock

This clock is driven by a signal derived from the CPU clock crystal oscillator. As this is subject to long term drift, consideration should be given to accuracy requirements when using it for critical timing applications. Typical accuracy in in the order of a minute or two per day.

As explained in section 1.2, user programs running in processor 1 may update the time-of-day clock by direct calls to processor 1's unique ROM. Two subroutine entry points, EXROUT and EXREST, are defined for this purpose.

EXROUT (entry point $F803) maintains a twelve-hour clock in an 8-character ASCII string called TIME, at locations $FE60 to $FE67. The format of this string is "HH:MM<sp>XM", where HH is the hours reading, MM is the minutes reading, and XM is either AM or PM.

The TIME string is updated once per minute provided EXROUT is called at least once every clock "tick" (every 15 ms minimum). Whenever a minute has been counted EXROUT stores a space character in place of the colon between the HH and MM fields. This is detected on the next call to EXROUT (about 15 ms later), which responds by resetting an internal seconds counter (at location $FE70), incrementing minutes, hours and AM/PM fields as necessary, and replacing the colon in the string.

The TIME string may be read through either processor. It should check that the colon is present to verify that the reading is stable. The TIME string may be initialised externally by first ensuring that EXROUT is called with a space in place of the colon, to reset the internal seconds counter, then storing the correct time in the ASCII string.

An 8-character DATE string is also defined at locations $FE68 to $FE6F. This is for external use only and is not maintained by EXROUT.

EXREST (entry point $F800) is a reset routine which is automatically executed on power-up or when the entire I/O system is to be reset. It initialises the PIA to receive time-of-day clock ticks and checks that the colon and "M" are present in the TIME string. If either is not, the string is set to "00:00 AM" and the DATE string is set to "00-00-00". This means that the TIME will not be lost by operating the Restart button, but it will be initialised after a power-on restart.

4.2.2 User Timer

The processor 1 clock routines also service a number of software timers for various system functions. One of these timers is reserved for use by the user program. This is a byte at $FE34 which is automatically decremented every "clock tick", which occurs about once every 15 milliseconds, until it reaches zero.

4.2.3 Keyboard Character Flag

It is sometimes desirable to be able to check if a character has arrived from the ACIA so that program control can be transfered accordingly. For this purpose the Keyboard Character Transfer Port can be read. This is a byte at $FE2A which is clear unless there is a character waiting, in which case the character can be fetched via the normal console input routines.

4.3 Floppy Disk Controller Subroutines

The floppy disk controller module firmware is used to control all of the diskette hardware functions. Parameters required by the firmware functions are stored in RAM in the locations described in the following table:

NAME	ADDRESS	DEFINITION
CURDRV	$0000	This byte contains the binary logical unit number of the drive to be selected (zero through three).
STRSCT	$0001	These two bytes contain the physical sector number of the first sector to be used (starting sector).
NUMSCT	$0003	These two bytes contain the number of sectors to be used. This number includes a partial sector, if a partial sector read is being requested. The sum of STRSCT and NUMSCT cannot be greater than $7D2 (single-sided diskettes) or $FA2 (double-sided diskettes).
LSCTLN	$0005	This byte contains the number of bytes to be read from the last sector during a read operation. This number should be a multiple of eight and cannot be greater than 128 ($80). If a number is specified that is not a multiple of eight, the next larger multiple of eight bytes will be read.
CURADR	$0006	These two bytes contain the first address in memory that is to be used during a read or write operation. This location is updated after each sector is read or written. During write test operations, these two bytes contain the address of a two-byte data buffer.
FDSTAT	$0008	This byte contains a status indication of the performed function. If an error occurred during a diskette operation, the carry bit in the condition code register will be set to one upon returning to the calling program. In addition, FDSTAT will contain a number indicating the error type ($31 - $39). These errors are explained in detail in the QDOS User's Guide. If no error occurs, then the carry bit of the condition code register will be set to zero and FDSTAT will contain the value $30.

For all of the firmware entry points described below, the contents of the registers is unspecified both upon entry and exit from the routine. Each entry point is accessed by executing a JSR or JMP instruction and returns via an RTS. The parameters must have been set up in RAM as indicated for each specific function.

NAME	ADDRESS	FUNCTION
OSLOAD	F800	This entry point initialises the drive electronics and loads the Bootblock and QDOS retrieval information block from the diskette in drive zero. The bootblock is given control after it has been loaded from the diskette. It, in turn, causes the rest of the operating system to be loaded into memory. No parameters are required for this entry point. This function does not return control to the calling program. If an error occurs during the Bootblock load process, the error number will be displayed on the console screen and control passed to the resident debug monitor. At least $120 bytes of memory are required starting at location zero. If less memory exists, the Bootblock program may not be able to display an error message indicating that there is insufficient memory in the system.
FDINIT	F803	This entry point initialises the diskette controller only.
CHKERR	F825	This entry point is used to check for a diskette controller error if called immediately after returning from another ROM entry point. The routine will check the state of the carry flag in the condition code register. If the carry flag is set to zero, the CHKERR routine will simply return to the calling program. if the carry flag is set to one (an error occurred), then the routine will print an "E" followed by the contents of FDSTAT and two spaces on the console screen. Control is given to the resident debug monitor after printing the error message. CHKERR does not change any of the parameters.
PRNTER	F82C	This entry point will print an "E" followed by the contents of FDSTAT followed by two spaces on the console screen. PRNTER does not change any of the parameters.

READSC	F806	This entry point causes the number of sectors contained in NUMSCT beginning with STRSCT from CURDRV to be read into memory starting at the address contained in CURADR. CURADR is updated to the next address that is to be written into after each sector is read. The parameter LSCTLN is automatically set to 128 ($80) so that a complete sector is read into memory when the last sector is processed. The parameters CURDRV, STRSCT, and NUMSCT are not changed. FDSTAT will contain the status of the read operation.
READPS	F80A	This entry point is similar to READSC with the exception that the last sector is only partially read according to the contents of LSCTLN. If LSCTLN contains 128 ($80), then this entry point is identical to READSC. The restrictions placed on LSCTLN are described in the preceding table of the parameters.
RDCRC	F80D	This entry point causes the number of sectors contained in NUMSCT beginning with STRSCT from CURDRV to be read to check their CRCs. The contents of the sectors are not read into memory. The only parameter changed is FDSTAT.
RWTEST	F810	This entry point causes the two bytes located at the address (and at address + 1) contained in CURADR to be written into alternating bytes of NUMSCT sectors beginning with STRSCT of CURDRV. After NUMSCT sectors are written in this fashion, they are read back to verify their CRCs. The only parameter changed is FDSTAT.
RESTOR	F813	This entry point causes the read/write head of CURDRV to be positioned to cylinder ZERO. The only parameter required is CURDRV. The only parameter changed is FDSTAT.
SEEK	F816	This entry point causes the read/write head of CURDRV to be positioned to the cylinder containing STRSCT. The only parameter changed is FDSTAT.
WRTEST	F819	This entry point causes the two bytes of data located at the address (and at address + 1) contained in CURADR to be written into alternating bytes of NUMSCT sectors beginning with STRSCT of CURDRV. The only parameter changed is FDSTAT.

WRDDAM F81C — This entry point causes a deleted data mark to be written to NUMSCT sectors beginning with STRSCT of CURDRV. The only parameter changed is FDSTST.

WRVERF F81F — This entry point causes NUMSCT sectors beginning at STRSCT of CURDRV to be written from memory starting at the address contained in CURADR. CURADR is updated to the address of the next byte to be read from memory after each sector is written. After all sectors have been written to the diskette, they are read back to verify their CRCs as checked by the routine RDCRC. The only parameters changed are CURADR and FDSTAT.

WRITSC F822 — This entry point is identical to WRVERF with the exception that the written sectors are not read back to verify their CRCs. The only parameters changed are CURADR and FDSTAT.

When an error occurs, the physical sector number at which the error occurred can be computed from the following relationship:

PSN = STRSCT + NUMSCT - SCTCNT -1

where PSN is the physical sector number at which the error occurred, and SCTCNT is a two-byte value contained in locations $000B-000C.

-SECTION 5-

5.0 DEBUG MONITOR PROGRAM (MONTR)

MONTR is a system program which serves as a user console driver, and as an aid to debugging other programs. The externally declared functions available are compatible with those provided by the Motorola programs EXBUG 1.1 and 1.2. ("EXBUG" is a registered trademark of Motorola corp.)

MON9 is a version of MONTR for the 6809. Unless otherwise stated below, all MONTR commands work identically for MON9. Some additional features are present in MON9 to handle long branches and the additional CPU registers of the 6809.

Via the input console, MONTR enables the user to:

-examine or alter the contents of any memory location,
-examine or alter the contents of any CPU register,
-examine @ an effective address, or offset, with return,
-calculate required branch offsets,
-determine the destination of existing branch offsets,
-fill a memory range with a specified value,
-establish program execution breakpoints,
-run a user program, setting CPU contents & start address,
-change between MONTR and MON9.

MONTR has the ability to operate breakpoints in the presence of a user SWI trap handler. No interference other than a moderate speed reduction is occasioned.

MONTR may also be called from the user program, without the establishment of a breakpoint. In this case, all facilities are available, including the ability to resume execution of the calling program.

An NMI interrupt may be used to call MONTR at any arbitrary point during execution of the user program. This is achieved by pressing the console interrupt for processor 2 on a dual 6800 QASAR or for processors 1 and 2 on a 6800/6809 system. As in the case of the program call, all of the MONTR facilities are available, including the ability to resume execution of the interrupted program from the point of interruption.

Section 5.9 of this manual contains a command summary for MONTR.

5.1 Command Prompt

When MONTR is started, or has returned to command input mode, a command prompt will be issued. The command prompt consists of a colon (:) on the left margin. MON9 prompts with an asterisk (*) on the left margin. With the prompt, either monitor is ready to accept user command inputs from the console keyboard.

Switching between MONTR and MON9 on a 6800/6809 system may be accomplished by typing M <return> in response to the command prompt. The new prompt symbol then indicates which monitor is active.

5.2 Opening commands

An open location is one whose contents MONTR has printed for examination and is then accessible for change. A closed location is no longer accessible. While a location is open, the user may enter a new value for the content of that location.

MONTR accepts several different commands, each having the effect of opening a memory location. The facilities offered by the commands differ as required by each function. These commands are explained in the following sections of this manual.

Note that in all examples MONTR's output is underlined.

5.2.1 The Slash, /

One way to open a location is to type its address, followed by a slash.

```
:3F8/2A
-    --
```

Location 3F8 (hex) is open, with content of 2A, and may be changed.

Used alone, the slash will reopen the last location previously open.

```
:3F8/2A<return>
-    --
:/2A
- --
```

As shown in the example, an open location may be closed by the return key (shown as <return>). In this case, location 3F8 was opened, closed by <return>, and reopened by another slash command.

5.2.2 The Reverse Slash, \ (shift-L)

Another way to open a location is to type its address with a reverse slash.

```
:3F8\2AF9
-   ----
```

Note in this example that MONTR has opened a 2-byte string, containing a 16-bit value. This 2-byte location is now open for examination and possible change as a single logical unit. This is the significant difference between the slash and reverse slash commands. The slash opens a single byte, while the reverse slash opens a 2-byte unit.

This function is particularly suited to examination and change of address values in memory.

As with the slash, the reverse slash will reopen the last location open.

```
:3F8/2A<return>
-   --
:\2AF9
- ----
```

also,

```
:3F8/2A\2AF9
-   -- ----
```

In these examples, 3F8 was opened as a single byte, then reopened as a 2-byte value.

When a 2-byte sequence is opened, the high order digits are taken from (address), and the low order digits are taken from (address+1). The first address of the pair may be odd or even, as required. No "word boundary" effect exists, as only byte strings are being used.

5.2.3 The Line Feed, <line feed> (control-J)

If a location is already open, a line feed will open the next in sequence.

```
:3F8/2A<line feed>
-   --
3F9/F9
------
```

also,

```
:3F8\2AF9<line feed>
-   ----
3FA\7B42
--------
```

Note that in the reverse slash case, line feed incremented the address by 2. The line feed command will take the next logical unit in sequence, whether 1 byte or 2 byte.

The current location will be closed, and optionally changed if a new value has been entered.

5.2.4 The Up Arrow, <up arrow> (shift-N)

If a location is already open, an "up arrow" character will open the previous location in sequence.

```
:3F9/F9<up arrow>
3F8/2A
```

also,

```
:3FA\7B42<up arrow>
3F8\2AF9
```

This operation is the reverse of the line feed. Note that in the reverse slash case, up arrow subtracted 2 from the open address value. The up arrow command will take the previous logical unit in sequence, whether 1 byte or 2 byte.

The current location will be closed, and optionally changed if a new value has been entered.

5.2.5 Named Locations, $

Several values are accessed by a name, instead of an address where the value is found. These include the CPU internal registers, after a breakpoint, and a small number of MONTR internal facilities.

The required syntax is $C, where C is a single character which has been assigned a valid meaning as a name. An open command is implied. the requested named value is opened, using a size of 1 or 2 bytes as required by the data.

After opening, a named value may be examined or changed as for an addressed location.

Defined named values are:

```
$A   CPU accumulator A
$B   CPU accumulator B
$D   CPU direct page register (6809 only)
$S   CPU Condition Codes
$X   CPU Index Register X
$Y   CPU index register Y (6809 only)
$P   CPU Program Counter
$$   CPU Stack Pointer S
$U   CPU Stack Pointer U (6809 only)
$O   Program Segment Relocation Register
$H   User SWI Handler address
$N   Terminal Null count.
```

Some examples of named values are:

```
:$A 24<return>
-  ---
:$B 56<return>
-  ---
:$X 10F2<return>
-  -----
```

In these examples, CPU registers A,B, and X were examined, and closed without change, by <return>.

5.3 Changing the Contents of a Location

The content of an open location may be changed by entering the new content, followed by a modify-and-close command. These commands are <return>, <line feed>, <up arrow>, <at sign>, <right bracket>, and <left bracket>. The entered new content will be written to memory, and the open location closed. After this has occured, any specific action due to the command selected will begin.

If a new content value is entered in error, a control-X character (produced by holding down the "CTRL" key, while striking the "X" key) will abort the change. This must be done before any modify-and-close command is issued. When control-X is used, the open location will not be modified, and the command prompt will be reissued for the next command.

5.3.1 The Return, <return>

If a location is open, the content may be changed by entering a new value, followed by a <return>. This is the simplest way to close a location. <Return> produces no effect other than closing the location and optionally changing the content of the location.

If a new value has not been entered, the open location is not changed by the <return>.

```
:12A7/35 7D<return>
-    ---
:12A7/7D
-    --
```

In this example, location 12A7 was opened, and found to contain 35. A new content of 7D was entered, and the location was closed by <return>. To check this, location 12A7 was reopened, and the content was found to be 7D.

```
:12A7/7D 5A<control X>
-    ---
:/7D 5B<return>
- ---
:12A7/5B
-    --
```

In this example, a mistake was made in the first attempt to change the location. <Control X> was used to cancel the error. The previously open location (12A7) was reopened, and the correct new content of 5B was entered. On opening 12A7 again to check, the new content was found to be 5B.

```
:$A FE 55<return>
-  ---
:$A 55
-  ---
```

In this example, a named value (CPU accumulator A) was opened. A new content was entered, followed by <return>. On checking, the new value is now found in $A.

5.3.2 Mixed Open, Modify and Close commands

MONTR commands may be strung together in arbitrary useful sequences. Those sequences involving open, examine, optional change and close are among the most common and useful in operations with MONTR.

Here is an example of a typical open and change sequence.

```
:1274\1136 5<line feed>
-    -----
:1276\1442 1339<return>
-----------
:\1339<up arrow>
- ----
:1274\0005
---------
```

In this example, a 2 byte unit was opened at address 1275. A new 2-byte content was entered, and the location was closed by <line feed>. <Line feed> also opened the next 2-byte unit in sequence, location 1276. Note that MONTR supplied leading Ø's as required.

At location 1276, a new 2-byte content was entered, and the location was closed by <return>. MONTR then issued a command prompt, and waited for a new command.

The new command was "\", which reopened the last open location (1276). This location is now seen to contain the new value. An <up arrow> then steps back to the original location (1274), where the new value is seen.

5.4 Address Sequence Operations

MONTR provides facilities for opening addresses in meaningful sequences. An address may be opened indirect (open a 2-byte value, use its content as an address, open that address). This is useful when considering program addresses in memory, or in going from the stack pointer to the stack content.

A facility exists to follow a branch address to its destination. A complementary facility exists for a return to the start of sequence. These are ready answers to the qestions "Where does this branch go ?" and "Where was I ?".

The return facility may also be used to facilitate a second look at a opened sequence from the beginning.

5.5.1 The "AT" Symbol, @ (shift-P)

The "at" symbol invokes indirect addressing, using the content of the currently open location as an address.

It is required that a 2-byte unit be open. The open location is closed, and optionally changed if a new value has been entered. After the optional change, the content is used as an address, which is opened.

```
:$$ FF86@
-   ----
FF86\1329
---------
```

In this example, the stack pointer was opened, then the "at" symbol was used to examine the top two bytes on the stack.

```
:142A\FFFF 1296@
-     -----
1296\24F3 <return>
----------
:142A\1296
-     ----
```

This example opened location 142A, entered a new content of 1296, closed 142A, and opened 1296. On return to 142A the new content of 1296 is seen.

5.5.2 The Right Angle Bracket, >

The right angle bracket function models the operation of a branch offset. It is required that a 1-byte unit be open. The open location is closed, and optionally changed if a new value has been entered. After the optional change, the content is used as an effective branch offset on the open location (address+1). The result of this calculation is used as an address, and that location is opened.

```
:1133/11>
-    ---
1145/86
-------
```

In this example, location 1133 was opened, and was found to contain a branch offset to the instruction at address 1145.

```
:1133/11 23>
-    ---
1157/86<return>
-------
:1133/23
-    ---
```

In this example, location 1133 was modified to contain the value 23. A branch offset of 23 at address 1133 was found to reach the instruction at address 1157. On reopening address 1133, the new value of 23 could be seen.

MON9 provides an additional facility to calculate long branch offsets for the 6809. If a 2-byte unit is open, the right angle bracket function uses the value as a long branch offset from the next location.

```
*1133\0120>
-    ----
1255\BD2F
---------
```

In this example, location 1133 was opened and found to contain a long branch offset to the instruction at address 1255.

5.5.3 The Left Angle Bracket, <

The left angle bracket commands a return to the start of the current sequence. On entry of slash, reverse slash, at symbol, or right angle bracket, the current address is stored. The left angle bracket command opens the location defined by the stored address.

This function provides the capability to return to the start of sequence, examined by line feed, or up arrow. It is also possible to operate using the at symbol, or right angle bracket (followed by line feed or up arrow commands as desired), and return to the starting point.

The current location will be closed, and optionally changed if a new value has been entered.

```
:1531/55>
-    --
1587/24<line feed>
-------
1588/12<
-------
1531/55
-------
```

In this example, a branch offset at 1531 was followed to 1587. While there, line feed was used to examine the next location. On seeing this, left angle bracket was used, returning to the start of this sequence at address 1531.

```
:2741\1144@
-     ----
1144\2436<up arrow>
---------
1142\FE23<
---------
2741\1144
---------
```

In this example, the address at 2741 was followed to 1145. While there, up arrow was used to examine the previous location. Then left angle bracket was used, returning to the start of this sequence at address 2741.

```
:1FB2/FF 86<line feed>
-     ---
1FB3/FF 41<
--------
1FB2/86<line feed>
--------
1FB3/41
-------
```

In this example, new values were entered at 1FB2 and 1FB3. Left angle bracket was then used to return to 1FB2 for a check of the new values.

MON9 performs the left bracket function in the same way, regardless of whether a long branch or short branch was followed.

5.5.4 Branch Offset Calculation, ;0

A relative branch involves the use of an offset representing the number of bytes forward or backward to the target address. When it is necessary to enter a branch address, MONTR will calculate the value required.

To calculate a branch offset from the currently open byte, enter the target address value, followed by <semicolon>, 0. The desired offset will be printed out, and the location reopened for entry.

```
:1471/FF 1491;0 1F
-     ---       ---
1471/FF 1F<return>
--------
```

In this example 1471 was opened, and the offset to 1491 was requested. When the location was reopened, the calculated offset was entered.

If the requested target address is out of branch range, MONTR will reply with "R?", and will return a command prompt.

```
:1471/FF 2491;O R?
-     ---      --
:
-
```

This example illustrates a range error. (2491 cannot be reached from 1471 with a 1-byte branch offset.)

MON9 performs this function in the same way when calculating short branch offsets. However, if a double byte value is open (using \) MON9 will calculate a long branch offset.

5.5 Breakpoint Facilities

MONTR provides breakpoint control for user programs. When a breakpoint is reached, MONTR regains control. While at the breakpoint, all program variables may be examined, or modified. If satisfactory, the program may be permitted to proceed, or it may be modified and restarted.

MONTR will maintain up to eight breakpoints at the same time, if needed. Assignment and release of a particular breakpoint is automatic, using the ;B and ;D commands.

All current breakpoints may be listed, by using the ;L command. All current breakpoints may be cleared, by using the ;C command.

MONTR uses the SWI instruction as a breakpoint call. A user SWI handler is permitted, and will run normally while debugging is in progress. To achieve this, $H must be loaded with the user SWI handler entry address. The user program must not reload the SWI vector, or MONTR will lose control.

5.5.1 Setting Breakpoints

To set a breakpoint, enter the desired address, followed by ;B. MONTR will record the requested address as an active breakpoint. When the user program is run under MONTR, SWI instructions are inserted in the program at all the active addresses. Breakpoints must be located on instruction addresses so that the SWIs will be executed. Note that a breakpoint cannot be set in ROM using this technique.

Up to eight breakpoints may be active at once. An attempt to set a further breakpoint after the eight have been used will be refused, and an error prompt will be issued.

To maintain EXBUG compatibility, it is not possible to set a breakpoint at address 0000.

```
:1372;B
-
:
-
```

In this example, a breakpoint was set at address 1372.

5.5.2 Removing Breakpoints

Breakpoints may be removed in two ways. When program execution is interrupted at a breakpoint, that breakpoint may be deleted by the ;D command. All existing breakpoints may be cleared at once by the ;C command.

```
:1249;B
-
:1243;B
-
:1236;G
-
B;1243
------
:;D
-
```

In this example, breakpoints were set at addresses 1249 and 1243. When the program was run, a breakpoint occured at address 1243, and MONTR regained control. The breakpoint at 1243 was deleted by the ;D command. The breakpoint at 1249 was not affected by this use of ;D.

```
:;C
-
:
-
```

In this example, all breakpoints are removed from the user program.

5.5.3 Examining Breakpoints

The ;L command will list all eight breakpoints. If a breakpoint is not in use, it will contain the value 0000.

```
:;L 1249 1243 0000 0000 0000 0000 0000 0000
-   ---------------------------------------
```

In this example, breakpoints are recorded as set at addresses 1249 and 1243. The remaining breakpoints were unused.

5.6 Running A Program

A user program may be started, or continued from a breakpoint by the use of the ;G and ;P commands.

5.6.1 The GO command, ;G

If an address is entered, followed by ;G, program execution will begin at the entered address. Before starting, the CPU will be loaded with the current values held under $A, $B, $X, $$, and $S (also $D, $Y, $U with MON9).

```
:1000;G
-
```

In this example, a user program was started at address 1000.

```
:$A 23 55<return>
- ---

:521;G
-
```

In this example, accumulator A was initialized with the value hex 55 before starting the user program at location 521.

5.6.2 The Proceed command, ;P

The proceed command may only be used after a breakpoint or "abort" (via the console interrupt button).

The effect of ;P is to continue the interrupted program from the apparent point of interruption. CPU register contents may be modified if desired. If CPU register contents are modified, program execution will continue with the new content.

To permit execution of the current instruction, any breakpoint at the current instruction location is temporarily removed from the program. This breakpoint will not be reinserted until MONTR regains control from another breakpoint, an abort, or monitor call.

A single breakpoint in a loop will be executed once only, then the loop will run to completion. To observe the progress of a loop, two or more breakpoints must be present within the loop.

If only a single breakpoint exists within a program, and ;P is used without inserting another breakpoint, MONTR will not regain control unless an abort or monitor call occurs.

```
B;126A
```

```
------
:;P
-
```

In this example, a breakpoint occured at location 126A. The ;P command was used to continue user program execution.

The breakpoint at 126A is now inactive, and will remain inactive until MONTR regains control via another breakpoint, abort, or monitor call.

```
B;126A
------
:;P
-
B;1274
------
:;P
-
B;126A
------
:;P
-
B;1274
------
:
-
```

In this example, a loop exists, containing two breakpoints. The two breakpoints will occur alternately until the loop terminates.

5.7 The FILL command, ;F

The fill command will load memory from a selected start address up to and including a selected end address. The value loaded is the same in all bytes, and is specified in the command.

The required command syntax is VV;F, where VV is the single byte hex value to be set into the specified memory locations.

```
: 55;F
-
BEG ADDR 1247<return>
---------
END ADDR 2147<return>
---------

:
-
```

In this example, memory locations 1247 through 2147

inclusive were set to the value 55.

The default value for VV is Ø. A memory clear operation requires only ;F, and desired address range.

The FILL command is not implemented in MON9 but may be performed from that monitor by first switching to MONTR with the "M" command.

5.8 Address Relocation Techniques

When using a relocating assembler and linking loader, difficulties often arise in determining physical addresses in memory. Relocatable assembly listings assume that each segment begins at address ØØØØ. The linking loader will then give a linked base address for each segment as loaded. To find a program address, the listing address must be added to the linkage base address. If done manually, this addition process is error prone and very tedious.

MONTR provides a means of operating from the assembled address value. The base address given from the link proceedure can be added to the assembled address by the MONTR relocation register $0, and the "." command.

A similar facility is provided to simulate the indexed addressing mode. The current index register content can be used to index on an entered address value by the "," command.

5.8.1 The Segment Relocation Command, "."

The "." command is applied to a previously entered address value. It has the effect of adding the current content of the relocation register ($0) to the entered value.

To use this facility, first open the relocation register, and enter the base address of the program segment required. After this is done, addresses may be entered from the assembly listing, and relocated to the linkage base by the "." command.

```
:$0 1244 3646<return>
-   -----

:141./86<line feed>
-    --
3788/41
-------
```

In this example, a segment base address of 3646 was set in the relocation register. Using this value as the segment base, address 141 plus the base was opened. This was done by entering "141", followed by ".", then followed by the opening command. The sequence "141", "." was equivalent to directly entering the sum of 3646 and 141. When line feed was used, the absolute open address value could be seen.

```
:11.\2742<up arrow>
-    ----
3655\2213
---------
```

In this example, the relocation register is assumed to still contain 3646. Using this base address, 11 plus base was opened, using "11" followed by "." and "\". When up arrow was used, the absolute address could be seen.

5.8.2 The Index Relocation Command, ","

The "," command is applied to a previously entered value. It has the effect of adding the current content of the index register (as saved in $X) to the entered value. A major use of this facility is to model the indexed addressing mode.

```
:$X 1002<return>
-   ----
:11,/29<return>
-    --
```

In this example, the index register contains 1002. An indexed address byte of value 11 will reach a data value of 29. The actual address referenced will be 1013, or the sum of $X and the entered value.

Note that this facility is configured for convienient operation on indexed address operations, during program operation. The value in $X is in fact the current index value.

Do not arbitrarily alter $X to use this function if a proceed command is considered, as the original index register content will be lost.

5.9 MONTR Command Summary

This is a short summary of commands and facilities existing in MONTR. For a full explanation of any command, refer to the appropriate section.

```
M      Switch to other monitor (MON9 or MONTR)
AAAA.  Relocate address AAAA by Relocation Register $0
B,     Index on the value B with index register $X
AAAA/  Open the 1-byte unit at address AAAA
/      Reopen the last open address as a 1-byte unit
AAAA\  Open the 2-byte unit beginning at address AAAA
\      Reopen the last open address as a 2-byte unit
$A     Open CPU accumulator A
$B     Open CPU accumulator B
$D     Open CPU Direct Page Register (6809 only)
$S     Open CPU Condition Code Register
$X     Open CPU index register X
$Y     Open CPU index register Y (6809 only)
$P     Open CPU Program Counter
$U     Open CPU User Stack Pointer U (6809 only)
$$     Open CPU Stack Pointer S
$0     Open program segment Reloction Register
$H     Open user SWI handler address (or bad entry facility)
$N     Open terminal null count

<return>          Close the open location
<line feed>       Close, open the next location
<up arrow>        Close, open the previous location
">"               Close, take branch offset and open
"@"               Close, take absolute address and open
"<"               Close, return to sequence start and open

AAAA;B    Insert a breakpoint at address AAAA
 ;L       Look at the current assignment of all breakpoints
 ;D       Delete the current breakpoint
 ;C       Clear all breakpoints

AAAA;G    Start a user program at address AAAA
 ;P       Proceed from breakpoint, abort, or call

AAAA;O    Calculate branch offset from open loc. to address A
VV;F      Fill memory from BEG ADDR to END ADDR with value VV

<control X>   Abort current command line, take no action.
```